XCSL: XML Constraint Specification Language

Marta H. Jacinto*
Universidade do Minho, Departamento de Informática
Braga, Portugal, 4710-057
Marta.jacinto@itij.mj.pt

Giovani R. Librelotto†
Universidade do Minho, Departamento de Informática
Braga, Portugal, 4710-057
grl@di.uminho.pt

José C. Ramalho and **Pedro R. Henriques**
Universidade do Minho, Departamento de Informática
Braga, Portugal, 4710-057
{jcr, prh}@di.uminho.pt

May 14, 2004

Abstract

After being able to mark-up text and validate its structure according to a document type specification, we may start thinking it would be natural to be able to validate some non-structural issues in the documents. This paper is to formally discuss semantic-related aspects. In that context, we introduce a domain specific language developed for such a purpose: XCSL. XCSL is not just a language, it is also a processing model. Furthermore, we discuss the general philosophy underlying the proposed approach, presenting the architecture of our semantic validation system, and we detail the respective processor. To illustrate the use of XCSL language and the subsequent processing, we present two case-studies. Nowadays, we can find some other languages to restrict XML documents to those semantically valid — namely Schematron and XML-Schema. So, before concluding the paper, we compare XCSL to those approaches.

Keywords: XML, Document Semantics, XCSL, XML-Schema, Schematron, Constraint Specification.

1 Introduction

In this paper we are concerned with the semantic specification of documents marked up in XML. Therefore, and to go straight to our target topic, we clearly assume that the reader is familiar with

*Currently working at ITIJ - Computer Department, Ministry of Justice, Portugal

†Work is supported by a grant from CNPq - Brazil

XML and companion for document's structuring and processing. For details on these topics we suggest the reading of [18].

XCSL, XML Constraint Specification Language, is a domain specific language with the purpose of allowing XML designers to restrict the content of XML documents. It is a simple, and small language tailored to write contextual conditions constraining the textual value of XML elements, in concrete documents (instances of some family specified by a DTD).

Basically, the language provides a set of constructors to define the actions to be taken by an XML semantic validator. These actions are triggered whenever a boolean expression, over the attributes' value or the elements content, evaluates to false. These predicates, that we call *constraint* or *contextual conditions* are also written in XCSL syntax. XCSL also designates a processor that traverses the document's internal representation (the tree) and checks its correctness from a semantic point of view (remember that the structural correctness is validated by the parser that builds the tree).

Moreover, XCSL is an XML language; so it becomes possible to add restrictions to XML documents using an XML dialect. That approach offers a complete XML framework to couple with syntax and semantics to document designers. The benefits of such an approach are obvious.

Constraining the content of each document's component is a way to guarantee the preservation of its semantic value; the need for semantic specification and its implications are discussed in section 2.

To keep the approach as standard as possible, XCSL conditions are translated into the XSLT [4] pattern language; this decision enables the use of standard XSL [13] processors to implement the validator. Those topics are discussed in section 3. The details concerning the implementation of the XCSL processor are introduced in section 5.

A formal specification of the proposed language, XCSL, is provided in section 4, showing a diagram that depicts the XML-Schema [11], and listing the respective DTD; in that section, we also detail the elements introduced in the DTD. For those more familiar with grammar based language definitions, we include the XCSL's CFG for as well.

To illustrate the claimed positive contributions of XCSL, we introduce (in section 6) two real life case-studies taken from a set of official documents produced in the context of Portuguese laws and another one for databases. We show these examples once they are expressive enough and complement the little examples that are shown along the whole document. Please notice, however, that the size of a document does not necessarily have to do with its semantic complexity — a huge document may, actually, have no semantic problems. Nevertheless, should you like to analyse more examples, they can be found in [7] and [8].

A synthesis of the paper and hints on future work are presented in the last part, section 8. Before that concluding section, we compare (in section 7) our proposal with related work; namely, Schematron [10] and XML-Schema are briefly introduced and then, through an example, we show similarities and main differences.

2 Document Semantics and Constraints

XML is aimed at document structure definition. At the present, XML, with the available DTDs, transformation languages and tools, constitute a strong and powerful specification environment.

However, within this framework, it is not possible to express constraints or invariants over the elements content, as to achieve that it is necessary to deal with documents static semantics.

Until now and concerning content constraining, we have just felt the need to restrict atomic element values, to check relations between elements or perform a lookup operation of some value in some database. This probably happens because other validations at higher levels are enforced by the XML parser according to a specific XML type definition (DTD or XML-Schema).

Therefore, semantic restrictions can be classified as belonging to one of the following categories:

Domain range checking This is the most common constraint. We need this type of constraint when we want a certain content/value to be between a pair of values (inside a certain domain). Normally, it is used when data is of type date or numeric. Example: *the price of a CD should be greater than X and less that Y.*

Dependencies between two document nodes There are cases where the value of an element/ attribute depends on (must be related someway with) the value of another element or attribute located in a different branch of the document tree. Example: *in Portuguese language, nouns and verbs must agree in person and number.*

Pattern matching against Regular Expression Sometimes we need to constrain (in any sense) a subset of words to follow a specific pattern. Example: *verify (lookup in a table, or something alike) all the words that appear in the source text inside inverted commas.*

Complex constraints We group in this category all the remaining constraints. However, there are two situations that appear with some frequency so we split this category in three subcategories:

Mixed content constraint When we want to enforce some cardinality and/or some order, for example over a mixed content. Example: *an element with mixed content must have certain sub-elements in a certain order and number in the middle of the free text.*

Singularity constraint When we want to enforce an uniqueness invariant over an element inside a certain context. Example: *the content of a certain element, in some context, shall be unique.*

Other constraints Whenever a constraint does not fit in any of the above categories or results from mixing some of the others. Example: *A poem must have two quatrains and two tercets exactly by this order, but only if it is of type sonnet.*

When going towards a semantic validation model we can distinguish two completely different steps:

- the definition – the syntactic part of the constraining model; the statements that express the constraints.
- the processing – the semantic part of the constraining model; the interpretation.

These two steps have different goals and correspond to different levels of difficulty at implementation time.

The definition step involves the specification of a new language or the adoption of an existent one; so we could just add extra syntax or design a new language that could be embedded in XML or coexist outside.

The first task in order to define a constraint specification language is to elaborate a detailed description of what we want to accomplish with that language: what kind of constraints do we want to be able to specify.

The XCSL language was conceived with all the four categories of restrictions enumerated in mind. Its current version, which we are focusing on in this paper, enables us to specify constraints belonging to any of these categories.

For the processing step, we need to create an engine with the capability to interpret the statements written in the above language, this is, to analyse them and evaluate the logical expression.

Static Semantics can be defined as a set of preconditions or contextual conditions over content. These contextual conditions could be checked by the XML parser during the abstract document tree building process or, if the user is using a structured editor, the built-in parser could check them during edition. This is not the approach followed in this work, because that solution requires that we remake a parser or create a new one, and our aim here is to make use of existing technology to implement our ideas, as much as possible. Our purpose is explained in the following section.

3 The XCSL approach

In [1], Ramalho wrote about his quest for an XML Constraint Specification Language (XCSL). The first questions raised were: Do we really need a new language? This new specification should be linked to elements or should be put inside the DTD?

The first idea was to specify the constraints together with the elements and attributes in the DTD. That would be a good solution if we intended to associate a constraint with an element. Nevertheless, we soon realized that we should associate constraints with context and not with elements: an element can appear in different contexts in a document tree and we may wish to enforce different constraints for each context. We needed a context selector like the one in query languages or style languages so the next step became the study of those languages.

Many languages were studied: XML query languages like XSLT [4], XQL [5], Element Sets [14], Lore [16], XML-GL [20], DSSSL [17], scripting languages like Perl[1] (with XML::Parser, XML::DT) and Omnimark[2].

From these, the scripting languages were discarded because we wanted a more user-friendly language, declarative and maintaining the good characteristics of XML, like hardware and software independence.

From text review, we easily concluded that XSLT was a common subset to most of the existing query languages. Besides that, XSLT has a feature that proved very useful to specify constraints: predicates.

Choosing XSLT as one of the core components of XCSL was a good strategic decision: this enabled us to use a standard XSL processor as the constraint validator engine.

[1] http://www.perl.com/

[2] http://www.omnimark.com/

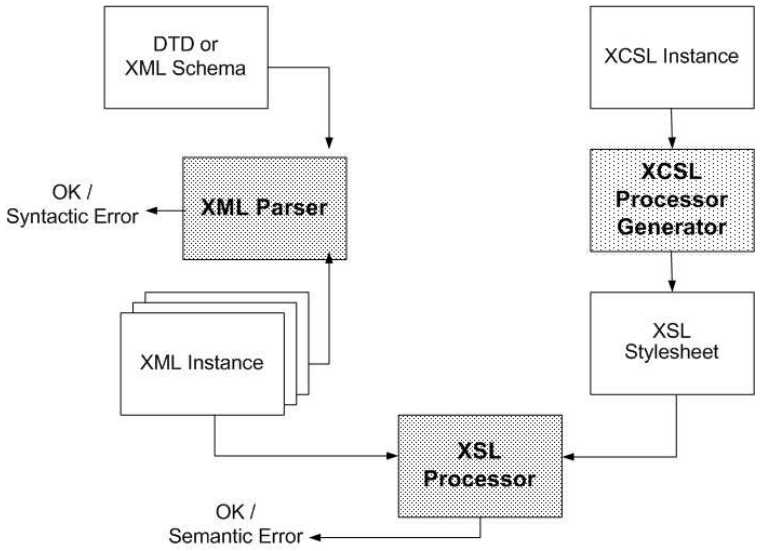

Figure 1: XCSL processing

So, we have defined a set of XML elements to wrap XSLT expressions. That way we built in the language that we called XCSL.

To finish this summary we exhibit an overview of the XCSL architecture. Figure 1 shows the XCSL workflow, the process to validate documents' semantics driven by XCSL Constraint Specifications: an XCSL document, describing the constraint to be validated, is given to the XCSL Processor Generator that produces an XSL stylesheet; then, using any standard XSL processor, it is possible to apply that stylesheet to the XML instance we want to check; and as a result we obtain another XML document with the error messages (the doc-status element will be empty if the source document is semantically correct). Figure 1 also describes the traditional structure validation process: the same XML instance, under check, is submitted to an XML parser with its DTD, and the Parser produces the syntactic error messages in a similar way.

The complete details about XCSL will be discussed in the following section.

4 The XCSL Language

Starting from an old idea to add semantics to SGML documents [19], the first version of the Constraint Specification Language is formally defined in [1] as it was already stated. After that very first exercise of formalization (that helped us to find the core structure of the language) we refined its definition and improved the processing engine to be able to couple with all practical problems we were faced to.

A specification in XCSL is composed by one or more tuples. Each tuple has three parts [6]:

- **Context Selector**: As the name suggests, this part contains the expression (a tree path) that selects the context where we want to enforce the constraint.

- **Context Condition**: This part corresponds to the condition we want to enforce, within the selected context.
- **Action**: The third part describes the action to trigger every time the condition does not hold.

In a more formal way, we can write the CFG (Context Free Grammar) for that language:

```
ConstraintSpec ::= Header Constraint+
Constraint ::= ContextSelector ContextCondition Action
Header ::= name date version
ContextSelector ::= XPath-Exp Let*
Let ::= Name Value
Value ::= XPath-Exp
ContextCondiction ::= RegExp | XPath-Exp
Action ::= message+
```

The choice of using XSLT to specify the constraints was already justified. In order to be coherent, we needed an XML wrapper for XSLT expressions (like in XSL). So, each XCSL specification is defined as an XML instance and the XCSL language is defined by a DTD and/or an XML-Schema; the present version of the DTD that specifies XCSL (named xcsl.dtd) is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- XCSL: XML Constraint Specification Language -->
    <!ELEMENT cs (constraint)+>
    <!ATTLIST cs
        dtd CDATA #IMPLIED
        date CDATA #IMPLIED
        version CDATA #IMPLIED
    >
    <!ELEMENT constraint (selector, let*, cc, action)>
    <!ELEMENT selector EMPTY>
    <!ATTLIST selector
        selexp CDATA #REQUIRED
    >
    <!ELEMENT let EMPTY>
    <!ATTLIST let
        name CDATA #REQUIRED
        value CDATA #REQUIRED
    >
    <!ELEMENT cc (#PCDATA | variable)*>
    <!ELEMENT variable EMPTY>
    <!ATTLIST variable
        selexp CDATA #REQUIRED
    >
    <!ELEMENT action (message*)>
    <!ELEMENT message (#PCDATA | value)*>
    <!ATTLIST message
        lang CDATA #IMPLIED
```

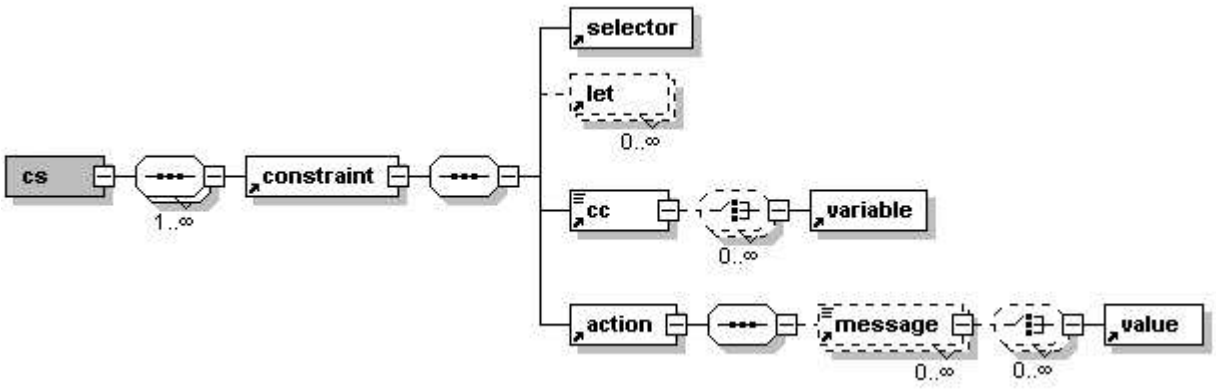

Figure 2: XCSL version 1.0 XML-Schema diagram

```
>
<!ELEMENT value EMPTY>
<!ATTLIST value
    selexp CDATA #REQUIRED
>
```

Nowadays, XML-Schema has overcome the DTD approach to the definition of classes of the XML documents. We also made that upgrade; however, as XML-Schema is much more verbose than the correspondent DTD, we decided to include just a diagrammatic description of the XCSL XML-Schema. This diagram is shown in Figure 2, as obtained with the XML Spy 4.1[3], from Altova.

The main elements introduced in the XCSL DTD above are briefly described in the next two paragraphs.

cs is the root element of an XML Constraint Specification Document. It has three attributes, all of them optional: *dtd* – name of DTD that is to be associated with the constraints being specified; *date* – XCSL document's date; *version* – XCSL document's version.

An XCSL constraint document consists of one or more constraint elements. Each constraint element specifies a constraint and the action that will be triggered when this constraint is not held. The constraint element is composed by a sequence of elements: a selector element, zero or more let elements, a cc element, and an action element. These elements are described separately in the following sub-sections.

4.1 Context Selector

The selector element, as the name suggests, selects the context (the element or elements) within which the conditions should be tested. It has a required attribute: *selexp* – this attribute should hold an XPath (XML Path Language[4]) expression that performs the selection.

```
<selector selexp="//iORDERS/itemsREG2"/>
```

[3]http://www.xmlspy.com/
[4]http://www.w3.org/TR/xpath

let elements have an important role in complex constraints, they allow us to specify multi-step evaluations, enabling the simplification of very complex constraints. With a let element we can save, in a variable (name) the result of an XPath expression applied to any XPath path, or still the set of values that belong to that context.

let has two required attributes: *name* – specifies the variable's name; *value* – XPath expression applied to the context or to any XPath path.

```
<let name="keyorderc" value="orderc"/>
```

selector and let are both empty elements, that is, the information is described in their attributes and the elements do not have any content (there is no text between the open and close tags).

4.2 Context Condition

cc is an element that specifies the constraint that has to be verified – regular expression or XPath function. The action will be triggered whenever that expression or function is evaluated to false. It has a mixed content – text in which *variable* elements can occur any number of times.

```
<cc> count(//iORDERS/itemsREG2[orderc=$keyorderc])=1 </cc>
```

The variable element is used when we are enforcing a constraint over an element or attribute and we want to guarantee that this element or attribute is evaluated only if present in the document instance. If the element or attribute in question is absent from the document, the *constraint* will not be evaluated. This element has an attribute; *selexp*, that specifies the XPath path in order to select it.

4.3 Action

action is a required element that specifies the messages to be returned whenever the evaluated expression (that describes the constraint under check) results in a false value. Its content is a sequence of message elements.

message has mixed content: text in which the value elements may occur any number of times. It has an optional attribute; *lang*, that identifies the language the message is written in (enabling various outputs for the same document).

Finally, the value element is used to include the value of some element, attribute or XPath expression applied to some element/attribute of the XML document, in the message body. It is an empty element. It has a required attribute; *selexp*, that contains the XPath path to the element or attribute which value we want to show or an XPath function applied either to the present context, either to another XPath path.

```
<action>
    <message>WARNING: orderc: <value selexp="orderc"/>
              is not unique! </message>
</action>
```

5 The XCSL Processor Generator

The XCSL processor generator is the main piece in our architecture as it can be seen in Figure 1. It takes an XML instance, written according to the XCSL language, and generates an XSL stylesheet that will test the specified constraints when processed by a standard XSL processor like Saxon[5] or Xalan[6].

The first versions of the generator were coded in Perl using an XML down-translation module called XML::DT [9]. The reason was that the task is quite complex and we needed an open tool with strong text processing capabilities. XML::DT is a perl module developed to process transformations over XML documents. It has some specific built-in operators and functions, but we can still use all the functionalities available in Perl.

During the development of this generator we found some problems that had a strong impact in the final algorithm. The most important were:

Optional or non-filled elements – suppose you have specified a constraint for every element named X; suppose that element X is optional and in certain parts of the instance the user did not insert it; the system will trigger the action for every non-existent element X (the absence of an element that is part of a condition will make that condition always false).

Ambiguity in context selection - until now, we have just said that an XCSL specification is composed by a set of constraints; we did not say that these constraints are disjoint in terms of context. In some cases there is a certain overlap between the contexts of different conditions. This overlap will cause an error when transposed directly to XSL once a single specification of restrictions may have various constraints, but XSL processors can only match one context at a time. Each restriction involves a certain group of nodes from the abstract tree. In a whole, several restrictions deal with several groups of nodes not necessarily disjoined. This means that the validator may need to access some nodes of the abstract tree more than once. Each restriction is implemented in XSL by using a template, however, each template in a single stylesheet has to access different nodes. The solution to this problem is to place each validation in a distinct traversal, what can be done in XSL running each constraint in a different mode (in XSL each mode corresponds to a different traversal of the document tree). Another way of solving the problem would be creating a stylesheet for each constraint and afterwards generating a kind of makefile to execute them one after another.

After some iterations and after solving the enumerated problems, the main algorithm is now:

1. Convert each *constraint* element into an *xsl:template*
2. Use attribute *selexp* for the *xsl:template*'s *match* attribute
3. Convert each "stamped" path (*variable* element) into a predicate *[...]*, inside the *match* attribute
4. Convert each *let* element into an *xsl:variable*
5. Convert *cc* element into an *xsl:if* element (the *test* attribute of the *xsl:if* element is filled with the negation of *cc*'s content).

[5] http://saxon.sourceforge.net/
[6] http://xml.apache.org/xalan-j/

6. Put *message* contents inside template body converting: - each *value* element into *xsl:value-of*
7. Filter all remaining text nodes

Recently we have been developing the XSL version of the generator. The main function which generates a template for each constraint looks like the following:

```
<xsl:template match="constraint">
  <xsl:variable name="cc">
    <xsl:apply-templates select="cc"/>
  </xsl:variable>
  <xsl:variable name="sel" select="selector/@selexp"/>
  <xsl:variable name="pred">
    <xsl:apply-templates select="cc/variable" mode="pred"/>
  </xsl:variable>
  <xsl:comment>
  ..................NEW CONSTRAINT.....................
  </xsl:comment>
  <my:template mode="constraint{count(preceding-sibling::*)+1}"
                    match="{$sel}{$pred}">
    <my:if test="not({$cc})">
      <err-message>
        <xsl:apply-templates select="action"/>
      </err-message>
    </my:if>
  </my:template>
  <my:template match="text()" priority="-1"
              mode="constraint{count(preceding-sibling::*)+1}">
     <!-- strip characters -->
  </my:template>
</xsl:template>
```

Let us now go through a very simple example. Consider the following extract of an XML document that describes a CD in some database:

```
<?xml version="1.0"?>
<cd>
  ...
  <price>32.00</price>
  ...
</cd>
```

Further, we want to ensure that the price of every CD is within a certain range (0 and 100). In order to do that, we specify the following constraint in XCSL:

```
<?xml version="1.0"?>
<cs>
  <constraint>
      <selector selexp="/cd/price"/>
      <cc>(.>0) and (.<100)</cc>
```

```
        <action>
            <message>Price out of range!</message>
        </action>
    </constraint>
  </cs>
```

This semantic specification document will be processed by the XCSL Processor Generator, and the following XSL stylesheet will then be generated:

```
<xsl:stylesheet version="1.0">
  <xsl:template match="/">
    <doc-status>
      <xsl:apply-templates mode="constraint1"/>
    </doc-status>
  </xsl:template>
<!--..................NEW CONSTRAINT.....................-->
  <xsl:template mode="constraint1" match="/cd/price">
    <xsl:if test="not((.&gt;0) and (.&lt;100))">
      <err-message>Price out of range!</err-message>
    </xsl:if>
  </xsl:template>

  <xsl:template match="text()" priority="-1" mode="constraint1"/>
  <xsl:template match="text()" priority="-1"/>
</xsl:stylesheet>
```

This stylesheet, when applied to an XML source document by an XSL processor, will emit error messages whenever the constraint does not evaluate to true.

6 Case Studies

In this section we show two real life case studies taken from a set of examples of official documents produced in the context of Portuguese laws. That set of legal documents was chosen to make a comparative study between XCSL and alternative approaches, as it will be discussed in the next section.

The first case-study is the Fiscal Certificate Request and the second one is the Second Conference for a divorce. In the first one the constraints are linear but in the second we have to use a special function in order to accomplish our task.

Afterwards we show how we could specify and restrict a database.

6.1 Fiscal Certificate

Let us suppose that someone in Portugal asks for a fiscal certificate, which is a certificate of the goods declared by someone's relatives by the time of his or her death (this is compulsory so that the goods can be inherited). That document should include the identification of the dead one and both the dates: the one of that person's death and the certificate requirement's one; which are required fields. Obviously, the current date shall be the most recent. Moreover, people must ask

for this kind of certificate in the finance department of the last residence area of the dead one, so we should enforce that congruence too. The correspondent DTD for such a document, could be:

```
<!ELEMENT fcert (header, body, ending)>
<!ELEMENT header (#PCDATA | department)*>
<!ELEMENT department (#PCDATA)>
<!ATTLIST department
    place CDATA "0101">
<!ELEMENT body (requester, request)>
<!ELEMENT requester (#PCDATA | name | CF | address)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT CF (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT request (#PCDATA | affinity | name | date | village | parish |
municipality)*>
<!ELEMENT affinity (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ATTLIST date
    value CDATA "19000101">
<!ELEMENT village (#PCDATA)>
<!ELEMENT parish (#PCDATA)>
<!ATTLIST parish
    place CDATA "010101">
<!ELEMENT municipality (#PCDATA)>
<!ATTLIST municipality
    place CDATA "0101">
<!ELEMENT ending (#PCDATA | place | date)*>
<!ELEMENT place (#PCDATA)>
```

The following XML instance is valid and illustrates a marked up *fiscal certificate request* according to Portuguese laws:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE fcert SYSTEM "fcert_cm.dtd">
<fcert>
  <header>
      Dear Sir, Chief of the Finance Department of
      <department place="110504">Lisbon's 4th Fiscal Parish</department>
  </header>
  <body>
      <requester>
          <name>Rita Santos </name>
          taxpayer Ner.
          <CF>31988455</CF>
          with the address
          <address>Pedras tortas Street, Ner 7 - 5423 Ranholas
        </address>
      </requester>
      <request>
```

```
            requests your Excellency to certify if, on behalf of the death of her
            ...
            <name>Francelestina Pereira e Santos</name>
            who died on the
            <date value="19990913">13th of September 1999</date>
            ...
            parish of
            <parish place="100611">Salir de Matos</parish>
            municipality of
            <municipality place="1006">Caldas da Rainha</municipality>
            and married she was with
            ...
        </request>
    </body>
    <ending>
        Ask that her request be granted
        <place>Caldas da Rainha</place>
        <date value="19991020">20th of October 1999</date>
        The requester
    </ending>
  </fcert>
```

This particular XML is valid from a static point of view. Any XML parser is able to check its structure against the given DTD. However if the current date was before the death's date, the document would be invalid and the DTD doesn't provide the means to verify that. Moreover, the finance department in which the request is being delivered is different from the last residence area's one and the document is still valid.

In order to be able to specify the first constraint, we added an attribute *value* to the *date* element which will keep the date in a standard format "yyyymmdd". The semantic constraint in XCSL is the following (we only show the *constraint* element as the rest is trivial)

```
 <constraint>
    <selector selexp="//request/date"/>
    <cc>
        @value &lt; /fcert/ending/date/@value
    </cc>
    <action>
        <message>
            The date of the death pointed out:
            <value selexp="/fcert/body/request/date"/>,
            is posterior to the request date:
            <value selexp="/fcert/ending/date"/>
        </message>
    </action>
 </constraint>
```

We compare the value of those two attributes: the first belonging to the *body* sub-tree (date of death) and the other one belonging to the *ending* sub-tree (date of the request). As the date

19990913 occurred before 19991020, the XML instance would be correctly validated. If this was not the case, an error message would be emitted; for instance, if the first attribute value was 20010803 and the second one was 20010607, the following error message would be triggered (as no language identification was specified in the *message* element, the output will be always this one):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc-status>
   <err-message>
       The date of the death pointed out: 3rd of August 2001,
       is posterior to the request date:  7th of June 2001
   </err-message>
</doc-status>
```

In order to be sure that the request is delivered in the appropriate Finance Department, we specified the following constraint:

```
<constraint>
   <selector selexp="//fcert/body/request"/>
   <cc>
       parish/@place = /fcert/header/department/@place
       or
       municipality/@place = /fcert/header/department/@place
   </cc>
   <action>
       <message>
           The request for this certificate shall not be delivered in this
           department <value selexp="/fcert/header/department"/>, but in
           the department in charge of the
           <value selexp="parish"/>'s parish,
           <value selexp="municipality"/>'s municipality.
       </message>
   </action>
</constraint>
```

Where we want to ensure that the *place* attribute of the *department* element is equal to the same *place* attribute of one of the elements *parish* or *municipality*.

To define such a constraint, we compare the value of each *place* attribute belonging to the *body* sub-tree (simplified by writing just the *request* element, descendant of the *body* element, once the first one is unique in the whole document) — with the *place* attribute belonging to the *header* sub-tree (place of the Department). As in this case the person is willing to deliver the request in a department which *local* attribute is not equal to the *parish*'s one nor to the *municipality*'s one, the following error message would be displayed (again, as no language identification was specified in the *message* element, the output will be always this one):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc-status>
   <err-message>
       The request for this certificate shall not be delivered in this department
```

```
        Lisbon's 4th Fiscal Parish, but in the department in charge of the
        Salir de Matos's parish, Caldas da Rainha's municipality.
    </err-message>
 </doc-status>
```

Moreover, when using DTDs we can not enforce order nor cardinality of elements allowed to occur in mixed content. Therefore, we will need a set of constraints that enforce both the order and cardinality inside the mixed content of the elements *header*, *requester*, *request* and *ending*.

Concerning the *requester* element, we want to force each sub-element to appear only once and the order of appearance to be: *name*, *CF* (tax payer number) and *address*. The respective constraint is:

```
 <constraint>
    <selector selexp="//fcert/body/requester"/>
    <cc>
        (count(name) = 1) and
        (count(CF) = 1) and
        (count(address) = 1) and
        name(name[1]/following::*) ='CF' and
        name(CF[1]/following::*) ='address'
    </cc>
    <action>
        <message>
            Either -requester- sub-elements occur in a wrong order,
            either they occur a wrong number of times.
        </message>
    </action>
 </constraint>
```

The number of occurrences of each of the elements *name*, *CF* and *address*, sub-elements of the *requester* element is counted and compared to 1. Besides, we verify that the *name* element is followed by a *CF* element, followed by an *address* element. The instance we presented would produce no errors, given that every element occurs only once and in the expected order.

The constraints for the other three elements are similar, so we do not show them here.

6.2 Second Conference for a divorce

In Portugal, to get divorced, a couple has to deliver two documents in court. The first one is called the *first request for divorce*, and is written when they decide to state their will to get divorced, assuring that in the presence of the judge. The second one, which we are focusing on in this example, is called the *second conference requirement* and the earlier it can be submitted is 90 days after the first one. This last one is to ask for a new meeting where the couple will state they still wish to get divorced, after which the couple will be officially divorced. At the moment of designing the DTD for this family of documents, we have two options: the number of days passed since the first conference is directly stated; or just the date of the first conference is written. It makes much more sense adopting the second one. Therefore, the DTD will be:

```
<!ELEMENT div_2c (header, body, ending)>
<!ELEMENT header (sender, addressee)>
<!ELEMENT sender (#PCDATA | cdepart)*>
<!ELEMENT cdepart (#PCDATA)>
<!ELEMENT addressee (#PCDATA | court)*>
<!ELEMENT court (#PCDATA)>
<!ELEMENT body (requesters, request)>
<!ELEMENT requesters (#PCDATA | name)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT request (#PCDATA | date | article)*>
<!ELEMENT date (#PCDATA)>
<!ATTLIST date
    value CDATA "19000101"
>
<!ELEMENT article (#PCDATA)>
<!ELEMENT ending (text, place, date, signature, signature)>
<!ELEMENT place (#PCDATA)>
<!ELEMENT signature (#PCDATA)>
<!ELEMENT text (#PCDATA)>
```

The following XML instance is valid and illustrates a marked up *second conference requirement* according to Portuguese laws:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE div_2c SYSTEM "div_2c02.dtd">
<div_2c>
  <header>
    <sender>
      Action of Divorce, Proc. Ner 2001/99
      <cdepart>2nd Department</cdepart>
    </sender>
    <addressee>
      Most worthy doctor of Laws, Judge of the
      <court> Judicial Court of the District of
               Caldas da Rainha  </court>
    </addressee>
  </header>
  <body>
    <requesters>
      <name>MARIETA MENDES </name>
      and husband
      <name>RICARDINO MENDES</name>
    </requesters>
    <request>
      identified in the referred Action of Divorce
      official papers, having accomplished the first
      conference in the
      <date value="20010406">6th of April of 2001</date>
      and both maintaining their will to divorce, come,
```

```
      by this means to require to be convoked for the
      second conference, according to the
      <article>1423th article of the Code of Civil Law
      </article>
      in order to the definite divorce be decreed.
    </request>
  </body>
  <ending>
    <text> Ask that their request be granted </text>
    <place>Caldas da Rainha</place>
    <date value="20010506">6th of May of 2001</date>
    <signature>The first requester's Lawyer</signature>
    <signature>The second requester</signature>
  </ending>
</div_2c>
```

Structurally, the document will be validated by any standard XML parser, but that check should not be successful unless the time between the present date and the date of the first conference is greater or equal to 90 days. It is essential that the requirement for the second conference occurs at least 90 days after first conference. For that, we need to compare both the dates; out of this comparison, we shall get the number of days in which they differ. To achieve this, we may use a function [12] that receives a Gregorian date and returns the Julian day for each date. Afterwards, it is enough to subtract them to get the number of days. Finally, comparing that result with the number 90, we know exactly whether the document is semantically correct or not.

We specify this semantic constraint in XCSL as follows (we only show the *constraint* element, as the rest is trivial):

```
<constraint>
  <selector selexp="//div_2c"/>
  <let name="a" value="(floor((14-substring(ending/date/
                       @value,5,2)) div 12))"/>
  <let name="y" value="(substring(ending/date/@value,1,4)
                      + 4800 - $a)"/>
  <let name="m" value="(substring(ending/date/@value,5,2)
                      + 12 * $a - 3)"/>
  <let name="t" value="(substring(ending/date/@value,7,2)
                      + floor((153 * $m + 2) div 5) +
                      (365 * $y) + floor($y div 4) -
                      floor($y div 100) +
                      floor($y div 400) - 32045)"/>
  <let name="a2" value="(floor((14-substring(body/request/
                       date/@value,5,2)) div 12))"/>
  <let name="y2" value="(substring(body/request/date/
                       @value,1,4) + 4800 - $a2)"/>
  <let name="m2" value="(substring(body/request/date/
                       @value,5,2) + 12 * $a2 - 3)"/>
  <let name="t2" value="(substring(body/request/date/
                       @value,7,2)+floor((153*$m2+2)
```

```
                        div 5) + (365 * $y2) +
                        floor($y2 div 4) -
                        floor($y2 div 100) +
                        floor($y2 div 400) - 32045)"/>
  <cc>
    ($t - $t2) >= 90
  </cc>
  <action>
    <message lang="en">
       Only <value selexp="($t - $t2)"/> days undergone
       since the first conference...
       You will have to wait a little longer!!
    </message>
    <message lang="pt">
       Só passaram <value selexp="($t - $t2)"/> dias desde
       a primeira conferência...
       Têm que esperar mais algum tempo!!
    </message>
  </action>
</constraint>
```

We use 8 elements *let* as they provide the modularity needed to specify this constraint in less lines. The context is the root element (*div_2c*). The first four *let* elements are applied to the *ending* branch of the document's tree (where we can find the date of the requirement itself) — we use the names *a*, *y* and *m* for the intermediary calculations and, finally, *t* to keep the Julian day of that date. The second set of *let* elements is applied to the *body* branch of the document's tree (where we can find the date of the first petition) — we now use the names *a2*, *y2*, *m2* and *t2*, with the same meaning.

After defining all this variables, the *cc* element itself is as simple as subtracting *t2* out of *t* and comparing the result with 90.

To provide a personalized result (error message) in case of an invalid petition, we can simply use a *value* element inside *message* element. We use the variables *t* and *t2* defined previously, avoiding the repetition of all the code.

The XML instance we show is structurally correct but semantically not correct once the first date is the 6th of April and the second one the 6th of May, both of the year 2001. Therefore, while validating this document against the constraint document specified above, we would get the error message shown below (in English once this is the default language, used whenever no information is provided):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc-status>
   <err-message>
       Only 30 days undergone
       since the first conference...
       You will have to wait a little longer!!
   </err-message>
</doc-status>
```

where 30 is the number of days between the two dates, generated automatically according to the *value* element specified in the constraint.

The *action* element has two *message* sub-elements, meaning that we could ask the output to be given in both (*lang="all"*) the languages or in a particular one. For the message in Portuguese, it would be enough to specify *lang="pt"* while invoking the tool, receiving the output:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc-status>
   <err-message>
       Só passaram 30 dias desde
       a primeira conferência...
       Têm que esperar mais algum tempo!!
   </err-message>
</doc-status>
```

6.3 Database

One database is made up of various tables, each one having several registries consisting of fields. Each registry shall have a key, i.e., a field with an unique value among all the values in that table. This way we will need to assure the uniqueness of some field only in the table in which that is the key field. The type ID is available for attributes, but means that that attribute has to be unique in the whole document, and that is not what we need. Therefore, we will specify a DTD for this kind of documents without the use of ID type of attributes and afterwards we will specify one constraint to deal with the uniqueness problem. The generic DTD, usable for any number of tables with any number of fields each, will be:

```
<!ELEMENT DB (STRUCTURE, DATA)>
<!ELEMENT STRUCTURE (TABLE)+>
<!ELEMENT TABLE (COLUMNS, KEYS)>
<!ATTLIST TABLE
    NAME CDATA #REQUIRED
>
<!ELEMENT COLUMNS (COLUMN)+>
<!ELEMENT COLUMN EMPTY>
<!ATTLIST COLUMN
    NAME CDATA #REQUIRED
    TYPE CDATA #REQUIRED
    SIZE CDATA #REQUIRED
    NULL (yes | no) #REQUIRED
>
<!ELEMENT KEYS (PKEYS)>
<!ELEMENT PKEYS (PKEY)+>
<!ATTLIST PKEYS
    TYPE (simple | complex) #REQUIRED
>
<!ELEMENT PKEY EMPTY>
<!ATTLIST PKEY
```

```
    NAME CDATA #REQUIRED
>
<!ELEMENT DATA (items)+>
<!ELEMENT items (items-REG+)>
<!ATTLIST items
    NAME CDATA #REQUIRED
>
<!ELEMENT items-REG (FIELD)+>
<!ELEMENT FIELD (#PCDATA)>
<!ATTLIST FIELD
    name CDATA #REQUIRED
>
```

The following XML instance is an extract of an XML instance for a database with four tables (stocks, suppliers, clients and orders):

```
<?xml version="1.0"?>
<!DOCTYPE DB SYSTEM "dbml_g.dtd">
<DB>
  <STRUCTURE>
     <TABLE NAME="stocks">
       <COLUMNS>
         <COLUMN NAME="cprod" TYPE="nvarchar" SIZE="10" NULL="no"/>
          ...
         </COLUMNS>
       <KEYS>
         <PKEYS TYPE="simple">
            <PKEY NAME="cprod"/>
         </PKEYS>
       </KEYS>
     </TABLE>
     <TABLE NAME="suppliers">
        ...
     </TABLE>
     <TABLE NAME="clients">
       <COLUMNS>
         <COLUMN NAME="cclient" TYPE="nvarchar" SIZE="10" NULL="no"/>
         <COLUMN NAME="name" TYPE="nvarchar" SIZE="50" NULL="no"/>
         <COLUMN NAME="contact" TYPE="nvarchar" SIZE="10" NULL="no"/>
         <COLUMN NAME="account" TYPE="nvarchar" SIZE="10" NULL="no"/>
       </COLUMNS>
       <KEYS>
         <PKEYS TYPE="simple">
            <PKEY NAME="cclient"/>
         </PKEYS>
       </KEYS>
     </TABLE>
     <TABLE NAME="orders">
```

```
     <COLUMNS>
       <COLUMN NAME="corder" TYPE="nvarchar" SIZE="10" NULL="no"/>
       <COLUMN NAME="cprod" TYPE="nvarchar" SIZE="10" NULL="no"/>
       <COLUMN NAME="quant" TYPE="nvarchar" SIZE="10" NULL="no"/>
       <COLUMN NAME="cclient" TYPE="nvarchar" SIZE="10" NULL="no"/>
     </COLUMNS>
     <KEYS>
       <PKEYS TYPE="simple">
          <PKEY NAME="corder"/>
       </PKEYS>
     </KEYS>
   </TABLE>
</STRUCTURE>
<DATA>
  <items NAME="stocks">
    <items-REG>
       <FIELD name="cprod">a111</FIELD>
       <FIELD name="description">agros meio-gordo milk</FIELD>
       <FIELD name="quant">150</FIELD>
       <FIELD name="csup">f019</FIELD>
    </items-REG>
    <items-REG>
       <FIELD name="cprod">a111</FIELD>
       <FIELD name="description">ucal meio-gordo milk</FIELD>
       <FIELD name="quant">230</FIELD>
       <FIELD name="csup">f231</FIELD>
    </items-REG>
    <items-REG>
       <FIELD name="cprod">b112</FIELD>
       <FIELD name="description">ucal meio-gordo milk</FIELD>
       <FIELD name="quant">204</FIELD>
       <FIELD name="csup">f231</FIELD>
    </items-REG>
      ...
  </items>
  <items NAME="suppliers">
    <items-REG>
       <FIELD name="csup">f019</FIELD>
       <FIELD name="name">Agros, S.A.</FIELD>
       <FIELD name="address">Porto</FIELD>
    </items-REG>
      ...
  </items>
  <items NAME="clients">
    <items-REG>
       <FIELD name="cclient">c001</FIELD>
       <FIELD name="name">Corner's Cafe</FIELD>
       <FIELD name="contact">123456324</FIELD>
       <FIELD name="account">12345678901234567890 1</FIELD>
```

```
      </items-REG>
      <items-REG>
         <FIELD name="cclient">c002</FIELD>
         <FIELD name="name">Supermimo Supermarket</FIELD>
         <FIELD name="account">098765432109876543210</FIELD>
      </items-REG>
        ...
    </items>
    <items NAME="orders">
      <items-REG>
         <FIELD name="corder">o012001</FIELD>
         <FIELD name="cprod">a111</FIELD>
         <FIELD name="quantity">10</FIELD>
         <FIELD name="cclient">c001</FIELD>
      </items-REG>
      <items-REG>
         <FIELD name="corder">o072001</FIELD>
         <FIELD name="cprod">b112</FIELD>
         <FIELD name="quant">20</FIELD>
         <FIELD name="cclient">c002</FIELD>
      </items-REG>
        ...
    </items>
  </DATA>
</DB>
```

This is a valid XML instance, even if several FIELD elements have the values a111 and b112 (a111 is repeated in the stocks TABLE); the contact FIELD was forgotten in one of the clients TABLE's records; and the quantity FIELD was used instead of the correct one - quant - in one record of the orders TABLE. This means that we will need to enforce several constraints in order to have usable XML documents.

To ensure the uniqueness of the key of each table, i.e. *cprod* attribute is unique in the branch that refers to the stocks TABLE, *csup* to the suppliers TABLE, *cclient* to the clients TABLE and *corder* to the orders TABLE, we will need four constraints, one for each sub-tree. The following constraint is to validate the first table's key uniqueness in XCSL (for the other three tables the constraints are similar):

```
  <constraint>
    <selector selexp="//items[@NAME='stocks']/items-REG[FIELD[@name='cprod']]"/>
    <let name="keycprod" value="FIELD"/>
    <cc>count(//items[@NAME='stocks']/items-REG[FIELD[@name='cprod'] =
$keycprod]) = 1</cc>
    <action>
      <message>WARNING:
        cprod: <value selexp="FIELD"/> is not unique!</message>
    </action>
  </constraint>
```

We use a let element to keep in keycprod all the values that the element FIELD takes in the context //items[@NAME='stocks']/items-REG[FIELD[@name='cprod']] (i.e. for the branch items with *name* attribute equal to 'stocks', every occurrence of FIELD for which the *name* attribute is equal to 'cprod'). After that, we count how many times each instance of FIELD occurs in keycprod list.

As we pointed out in the beginning of the example, a111 is repeated in the stocks TABLE. The other repetitions won't be pointed out as they occur outside the items[@NAME='stocks'] sub-tree. Therefore, the following errors will be displayed:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc-status>
   <err-message>WARNING:
       cprod: a111 is not unique!</err-message>
   <err-message>WARNING:
       cprod: a111 is not unique!</err-message>
</doc-status>
```

These constraints are not enough to completely validate our documents. We also need to validate, for each table:

- that each and every field defined in the STRUCTURE sub-tree is used to instantiate the records in the DATA sub-tree - there are no fields unused nor used more than once
- that each and every record in the DATA sub-tree uses FIELDs identifiers defined in the STRUCTURE sub-tree.

For the third TABLE, clients, the first restriction is:

```
<constraint>
  <selector selexp="TABLE[@NAME='clients']/COLUMNS/COLUMN"/>
  <let name="tableclients" value="@NAME"/>
  <cc>
    (count(//items[@NAME='clients']/items-REG/FIELD[@name = $tableclients]) =
      count(//items[@NAME='clients']/items-REG))
  </cc>
  <action>
    <message>WARNING:
      The field <value selexp="$tableclients"/> was not used in every record of the
      "clients" table (or was used more than once in some record).
    </message>
  </action>
</constraint>
```

With the let element, we place in tableclients all the values the *NAME* attribute assumes in the context TABLE[@NAME='clients']/COLUMNS/COLUMN (values the *NAME* attribute assumes every time the COLUMN element occurs in the sub-tree TABLE[@NAME='clients']). After this, we count, for each FIELD element belonging to the sub-tree items[@NAME='clients'], how many times

the *name* attribute occurs in the previously defined list tableclients. We also count the number of records (items-REG) of the sub-tree items[@NAME='clients'], to assure that each field defined for the clients TABLE was used as many times as the number of records described for that TABLE. The action is triggered when these two values are not the same.

In the example we are using, one of the records of the clients TABLE does not have the contact FIELD, therefore, the following error would be displayed:

```
<err-message>WARNING:
     The field contact was not used in every record of the
     "clients" table (or was used more than once in some record).
   </err-message>
```

The constraints for the other three tables are similar. Provided we substitute clients by the name of each of the other tables, the construction of the new constraints is trivial.

For the fourth TABLE, orders, the second restriction is (again, the constraints for the other three tables are similar):

```
<constraint>
  <selector selexp="items[@NAME='orders']/items-REG/FIELD"/>
  <let name="tableorders2" value="@name"/>
  <cc>
      (count(//TABLE[@NAME='orders']/COLUMNS/COLUMN[@NAME = $tableorders2]) > 0)
  </cc>
  <action>
   <message>WARNING:
     The field <value selexp="$tableorders2"/> doesn't exist for this table: "orders".
   </message>
  </action>
</constraint>
```

With the let element, we place in tableorders2 the list of values the *name* attribute assumes in the context items[@NAME='orders']/items-REG/FIELD (values the *name* attribute assumes every time the FIELD element occurs in the sub-tree items[@NAME='orders']). After this, we count, for each COLUMN element belonging to the sub-tree TABLE[@NAME='orders'], the number of times the *NAME* attribute occurs in the previously defined list tableorders2. The action is triggered when this value is greater than zero, this is, the FIELD used in the sub-tree items[@NAME='orders']) was defined in the sub-tree TABLE[@NAME='orders'].

In the example we are using, one of the records of the stocks TABLE has a quantity FIELD instead of the quant FIELD that has been defined in the STRUCTURE sub-tree, therefore, the following errors would be displayed (notice the first error is generated by the first kind of constraint, only the last one is produced by the last constraint shown):

```
<err-message>WARNING:
     The field quant was not used in every record of the
     "orders" table (or was used more than once in some record).
</err-message>
<err-message>WARNING:
     The field quantity doesn't exist for this table: "orders".
</err-message>
```

With these three kinds of constraints for each table, the documents will be correctly validated.

7 Related Approaches

In this section, we compare XCSL with Schematron [10] and XML-Schema [11], two other semantic specification approaches developed meanwhile with similar aims, representing the only two known works from other authors comparable with ours. For this comparison we will use only the second case-study presented (the other one can be found on [8], where an extensive work of comparison between the three approaches from different authors is presented).

7.1 Constraining with Schematron

Schematron is an XML schema language that combines powerful validation capabilities with a simple syntax and implementation framework. At Schematron's design and specification time, there were several aims, from which we highlight: to promote natural language descriptions of validation failures; to allow a more human-readable answer as the validation result; aim for a short learning curve by layering on existing tools (XPath and XSLT); support workflow by providing a system which understands the phases through which a document passes in its lifecycle.

By time being, the only possibility of specifying in Schematron the constraint we specified in subsection 6.3 is writing the whole equation each time we need to use it. In Schematron we do not have the *let* statement, so we can not use intermediate variables; this is due to the fact that, in Schematron language, only XPath functions are available and the one used in XCSL, that allows variables to be instantiated, is an XSL function. The equivalent constraint in Schematron would, therefore, be,

```
<title>Request for the 2nd conference of divorce</title>
<diagnostics>
 <diagnostic id="01">
   Less than 90 days undergone since the first
   conference...
   You will have to wait a little longer!!
 </diagnostic>
</diagnostics>
<pattern name="Days since the First Conference">
 <rule context="//div_2c">
   <assert test="
   (((substring(ending/date/@value,7,2)+
   floor((153*(substring(ending/date/@value,5,2)+12*
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))-3)+2) div 5)+
   (365 * (substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12)))))+
   floor((substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))) div 4)-
```

```
    floor((substring(ending/date/@value,1,4)+4800-
      (floor((14-(substring(ending/date/@value,5,2)))
      div 12))) div 100)+
    floor((substring(ending/date/@value,1,4)+4800-
      (floor((14-(substring(ending/date/@value,5,2)))
      div 12))) div 400)-32045)
    - ...)
    &gt;= 90" diagnostics="01">
   </assert>
  </rule>
 </pattern>
```

Where we put "...", we mean that all the code for the evaluation of the Julian day is repeated, now for the branch *body*. We don't repeat it to keep the example as light as possible. Notice that in this approach we are not providing the personalized output that we did with XCSL. To do this, it would be necessary to repeat all the code listed above for the evaluation of both the dates. This is clearly a disadvantage once the number of lines we need to specify the Schematron's constraint is huge when compared with XCSL's one.

This does not mean that we can not use a *key* element in Schematron, but that it can not be used for variables. The key element can only be used to keep a set of values in a list against which we will be able to compare other values — as we did to solve the *singularity problem* in databases.

With Schematron we can not have several output languages.

Our experiments allowed us to say that Schematron is clearly more complex than XCSL, and even if it is true that the first one has some possibilities inexistent in the last one (like documentation or the ability of entitling the whole restrictions' document), it is also true that the over all effort to learn those facilities overweighs the advantage of using them.

7.2 Constraining with XML-Schema

XML-Schema was created once the syntax of DTDs fell short of the requirements of the XML users. The aims of the *W3C XML Schema Working Group* were to create a language that would be more expressive than DTDs and written in XML Syntax. In addition, it would also allow authors to place restrictions on the elements' content and attribute values in terms of primitive datatypes found in most languages. While using DTDs, to specify constraints, even if very simple, we need to use a constraining language (such as XCSL or Schematron) and, consequently, need two documents to completely validate an XML instance. Constraining with XML-Schema, on the other hand, means, for a particular set of constraints, using only one document to validate XML instances instead of using both a DTD and a constraint document. Unfortunately, the range of constraints we can validate with an XML-Schema is far from the set we specified above.

In the case-study of subsection 6.3, we had to decide whether to write the number of days passed since the first petition or the second conference request's date. We used the date (*date*) back then which was not the best choice concerning the XML-Schema validator; if we did choose to use the number of days, we would be able to specify the constraint by using just an XML-Schema. For that, rather than the *date* element, we have a *howmany* element, which is of type *tahowmany*:

```
<xs:simpleType name="tahowmany">
```

```
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="90"/>
    </xs:restriction>
</xs:simpleType>
```

This subtype is a subset of *integer* defined, by restriction, allowing just values greater than or equal to 90. As simple as this, we can specify all Domain Range checking constraints with XML-Schemas.

Besides this kind of constraints, XML-Schema also allows to specify the cardinality of an element (which was not possible with DTDs). However, other constraints that we can specify, and check, in XCSL and Schematron, are impossible to deal with in XML-Schema.

8 Conclusion

We have been benchmarking XCSL, Schematron and XML-Schema; its discussion is out of the scope of this paper. The results so far attained are presented and compared in another paper [7]. However, the referred experiments permit us to say that XCSL was approved: we succeeded in applying this approach to all the case studies, virtually representative of all possible cases.

It means that: on one hand, we were able to describe the constraints required by each problem in a direct, clear and simple way; on the other hand, the semantic validator could process every document successfully, that is: keeping silent when the constraints are satisfied; and detecting errors, reporting them properly, whenever the contextual conditions are broken.

XCSL can either be used as a stand-alone validating application or together with DTDs or XML-Schema (most frequent option). For instance when we do not need an XML-Schema, and just want to validate a particular invariant or context condition, it is advisable to use XCSL as a stand-alone application.

Some reasons to use XCSL are:

- XCSL is XML. So we can use all the available tools to manipulate and process XCSL specifications.
- XML-Schema or DTDs are not always necessary, a simple XCSL specification can solve the problem.

The lessons we took from this work can be summarized as:

- Do it simple
- Do it with existing technology

Future work will pursue in two directions:

1. The generation of Perl (XML::DT) instead of XSL. This will allow the use of Perl operators in Contextual Conditions (patterns and actions will be written in Perl) making the system more expressive and powerful.
2. The reverse engineering of the whole system: trying to find a suitable abstract representation for these constraints and the whole model (probably High Order Attribute Grammars).

References

[1] Ramalho, J.C.: Anotação Estrutural de Documentos e sua Semântica. Universidade do Minho - Portugal (2000)

[2] Knuth, D.: Semantics of Context Free Languages. In Mathematical Systems Theory Journal (1968)

[3] Henriques, P.R.: Atributos e Modularidade na Especificação de Linguagens Formais. Universidade do Minho - Portugal (1992)

[4] Robie, J., Lapp, J., Sach, D.: XSL Transformations (XSLT) - version 1.0. In http://www.w3.org/TR/1998/WD-xsl-19980818 (2000)

[5] Robie, J., Lapp, J., Sach, D.: XML Query Language (XQL). In QL'98 - The Query Language Workshop (1998)

[6] Ramalho, J.C., Henriques, P.R.: Constraining Content: Specification and Processing. In XML Europe'2001, Internationales Congress Centrum (ICC), Berlin, Germany (2001)

[7] Jacinto, M.H., Librelotto, G.R., Ramalho, J.C., Henriques, P.R.: Constraint Specification Languages: comparing XCSL, Schematron and XML-Schemas. In XML Europe'2002, Barcelona, Spain (2002)

[8] Jacinto, M.H.: Validação Semântica em documentos XML. Universidade do Minho - Portugal (2002)

[9] Ramalho, J.C., Almeida, J.J.: XML::DT - a Perl Down Translation Module. In XML Europe'99, Granada, Espanha (1999)

[10] Dodds, L.: Schematron: Validating XML Using XSLT. In XSLT UK Conference, Keble College, Oxford, England (2001)

[11] Duckett J., Griffin, O., Mohr, S., Norton, F., Stokes-Rees, I., Willians, K., Cagle, K., Ozu, N., Tennison, J.: Professional XML Schemas. Wrox Press (2001)

[12] Tondering, C.: Frequently Asked Questions about Calendars - Version 2.3. In http://www.tondering.dk/claus/calendar.html (2000)

[13] World Wide Web Consortium: Extensible Stylesheet Language (XSL) - Version 1.0. In http://www.w3.org/TR/xsl/ (2001)

[14] World Wide Web Consortium: Element Sets: A Minimal Basis for an XML Query Engine. In http://www.w3.org/TandS/QL/QL98/pp/sets.html (1998)

[15] World Wide Web Consortium: Extensible Markup Language (XML). In http://www.w3.org/XML (2001)

[16] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.: The Lorel Query Language for Semistructured Data. In International Journal on Digital Libraries, 1(1):68-88 (1997)

[17] Clark, J.: Document Style Semantics and Specification Language (DSSSL). In http://www.jclark.com/dsssl/ (2001)

[18] Harold, E.R., Means, W.S.: XML in a Nutshell. O'Reilly & Associates (2001)

[19] Herwijnen, E.: Practical SGML. Kluwer Academic Publishers (1994)

[20] Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., Tanca, L.: XML-GL: A Graphical Language for Querying and Reshaping XML Documents. In http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html (1998)